How To Write a Setuid Program

Matt Bishop

May, 1985

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 85.6

Research Institute for Advanced Computer Science

How To Write a Setuid Program

Matt Bishop

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

Setuid programs can pose a grave threat to UNIX systems because they explicitly violate the protection scheme designed into UNIX. However, setuid programs are often the only practical solution to problems of maintaining a fully functioning UNIX system. Because of this paradox, they are among the most difficult programs to write. This paper lists and discusses some simple rules for writing setuid programs that will decrease an attacker's ability to use such a program to compromise a UNIX system.

May 23, 1985

How To Write a Setuid Program

Matt Bishop

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

Introduction

A typical problem in systems programming is often posed as a problem of keeping records [ALEP71]. Suppose someone has written a program and wishes to keep a record of its use. This file, which we shall call the *history file*, must be writable by the program (so it can be kept up to date), but not by anyone else (so that the entries in it are accurate.) UNIX† solves this problem by providing two sets of identifications for processes. The first set, called the *real* user identification and group identification (or UID and GID, respectively), indicate the real user of the process. The second set, called the *effective* UID and GID, indicate what rights the process has, which may be, and often are, different from the real UID and GID. The protection mask of the file which, when executed, produces the process contains a bit which is called the *setuid* bit. (There is another such bit for the effective GID.) If that bit is not set, the effective UID of the process will be that of the person executing the file; but if the setuid bit is set (so the program runs in *setuid mode*), the effective UID will be that of the owner

†UNIX is a Trademark of Bell Laboratories.

of the file, not that of the person executing the file. In either case, the real UID and GID are those of the person executing the file. So if only the owner of the history file (who is the user with the same UID as the file) can write on it, the setuid bit of the file containing the program is turned on, and the UIDs of this file and the history file are the same, then when someone runs the program, that process can write into the history file.

These programs are called *setuid programs*, and exist to allow ordinary users to perform functions which they could not perform otherwise. Without them, many UNIX systems would be quite unusable. An example of a setuid program performing an essential function is a program which lists the active processes on a system with protected memory. Since memory is protected, normally only the privileged user *root* could scan memory to list these processes. However, this would prevent other users from keeping track of their jobs. As with the history file, the solution is to use a setuid program, with *root* privileges, to read memory and list the active processes.

Setuid programs introduce many security problems [TRUS80]. This paper describes how to write such programs to minimize these problems. The reader should bear in mind that on some systems, the mere existence of a setuid program introduces security holes; however, it is possible to eliminate the obvious ones.

In this paper, all references to the *UNIX Programmer's Manual* are to either the 4.2 Berkeley manual [UPM83] or to the System V manual [UPM84]. As usual, manual pages are indicated by following the italicized name with the sec-

tion number in parentheses.

I. Be as Restrictive as Possible in Choosing the UID

The basic rule of computer security is to minimize damage resulting from a break-in. For this reason, when creating a setuid program, it should be given the least dangerous UID possible. If, for example, game programs were setuid to *root*, and there were a way to get a shell with *root* privileges from within a game, the game player could compromise the entire computer system. It would be far safer to have a user called *games* and run the game programs setuid to that user. Then, if there were a way to get a shell from within a game, at worst it would be setuid to *games* and only game programs could be compromised.

II. Do Not Write Setuid Shell Scripts

The Berkeley 4.2 Distribution of UNIX allows shell scripts to be run with setuid permissions. To understand how this works, a brief explanation is in order.

This version of UNIX checks the first line of a shell script to see if it begins with the two characters '#!'. When such a shell script is executed, the rest of that line, up to the first 32 characters, is taken as the absolute path name of a command interpreter, which is then executed†. If the shell script is setuid, the setuid bits are applied to the command interpreter *before* execution.

Unfortunately, once one finds a setuid shell script, it is very easy either to obtain an *interactive* setuid shell, or to force the shell to execute any specified

† See execup(3) for details.

sequence of commands. This leaves the owner of the shell script open to a devastating attack. Under *no* circumstances should a setuid shell script *ever* exist on *any* system where security is a concern.

One way to avoid having a setuid shell script is to turn off the setuid bit on the shell script, and rather than calling the script directly, use the following program to call it:

```
/*
 * This is a simple program to run
 * a script as though it were setuid
 * to the owner of this program.  The
 * executable of this must be setuid
 * to the owner of the shell script.
 */
main(argc, argv)
int argc;
char **argv;
{
    /*
     * Replace the zeroth argument
     * with the path name of the
     * shell script.
     */
    argv[0] = SCRIPT_FULL_PATH_NAME;

    /*
     * Overlay the script.
     */
    (void) execv(argv[0], argv);

    /*
     * If it gets here, the script
     * did not run ...
     */
    perror(SCRIPT_FULL_PATH_NAME);
    exit(1);
}
```

In this program, SCRIPT_FULL_PATH_NAME is the full path name of the shell script; as the comments indicate, the executable generated by compiling this pro-

gram must be made setuid to the owner of the shell script. However, the shell script should **not** be setuid.

III. Do Not Use *creat*(2) for Locking

According to its manual page, "The mode given [*creat*] is arbitrary; it need not allow writing. This feature has been used ... by programs to construct a simple exclusive locking mechanism." In other words, one way to make a lock file is to *creat* a file with an unwritable mode (mode 000 is the most popular for this). Then, if another user tried to *creat* the same file, *creat* would fail, returning −1. For example:

```
/*
 * This is supposed to provide a reliable locking
 * mechanism for programs.
 */
lock(lock_file_name)
char *lock_file_name;       /* lock file */
{
    return(creat(lock_file_name, 0));
}
```

The only problem is that such a scheme does not work when at least one of the processes has *root*'s UID, because protection modes are ignored when the effective UID is that of *root*. Hence, *root* can overwrite the existing file regardless of its protection mode.

To do locking in a setuid program, it is best to use *link*(2). If a link to an already-existing file is attempted, *link* fails, even if the process doing the linking is a *root* process and the file is not owned by *root*. Here is an example of a locking routine that uses this:

```
/*
```

```
 * These routines provide a reliable locking
 * mechanism for processes regardless of what
 * user id they have or who owns them.
 */
#include <errno.h>
extern int errno;          /* error code */

/*
 * The locking routine: note you give it the lock
 * file name and an existing file name so this
 * routine can be used with processes creating
 * multiple locks on different file systems.
 * It returns 1 if the lock was successful.
 * 0 if the lock failed because some other process
 * locked this one out, and -1 if the attempt
 * failed for any other reason.
 */
lock(existing_file_name, lock_file_name)
char *existing_file_name;       /* name of existing file */
char *lock_file_name;           /* name of lock file */
{
     /*
      * Be sure existing_file_name exists
      * If it does, creat fails, so we ignore
      * the failure.
      */
     (void) creat(existing_file_name, 0);
     /*
      * Try to make the link
      */
     if (link(existing_file_name, lock_file_name) == 0)
          return(1);
     /*
      * Oops - it failed. Return the
      * appropriate code.
      */
     return(errno == EEXIST ? 0 : -1);
}

/*
 * The unlocking routine: it's what you would
 * expect.  It returns 1 if the unlock succeeded,
 * 0 if it failed because the lock file did not
 * exist, and -1 if it failed for any other
 * reason.
 */
unlock(lock_file_name)
```

```
char *lock_file_name:        /* name of the lock file */
{
    /*
     * Try to break the link
     */
    if (unlink(lock_file_name) == 0)
        return(1):
    /*
     * Oops – it failed. Return the
     * appropriate code.
     */
    return(errno == ENOENT ? 0 : -1):
}
```

Note that the *link* call requires that its first argument exist, that both existing_file_name and lock_file_name be in the same file system, and existing_file_name not be a directory. The above locking routine returns 1 if the locking attempt succeeds, 0 if it fails because another process has locked it out, and −1 if it fails for any other reason. A return value of −1 means there is some problem, such as being unable to create existing_file_name. Similarly, the unlocking routine returns 1 if the unlocking succeeds, 0 if it fails because no locking was done, and −1 if it fails for any other reason.

With 4.2 Berkeley UNIX, an alternative is to use the *flock*(2) system call, but this has disadvantages (specifically, it creates advisory locks only, and it is not portable to other versions of UNIX).

The issue of covert channels [LAMP73] also arises here; that is, information can be sent illicitly by controlling resources. However, this problem is much broader than the scope of this paper, so we shall pass over it.

IV. Catch All Signals

When a process created by running a setuid file dumps core, the core file has the same UID as the real UID of the process†. By setting *umask*s‡ properly, it is possible to obtain a world-writable file owned by someone else. On some UNIX systems, a shell can be made to execute commands entered in that file with the rights of the owner of the file.

To prevent this, setuid programs should catch all signals possible§. If *init-sig*, defined below, is called on initialization, any signal will cause an immediate exit without a core dump:

```
/*
 * This catches all signals and exits
 * without dumping core.
 */
#include <signal.h>

/*
 * This just exits.  Since the signal number
 * is just the first argument to this routine.
 * you can get fancy if you want.
 */
catcher()
{
    exit(1);
}

/*
```

† On some versions of UNIX, such as 4.2BSD, no core file is produced if the owner of the setuid process is *root*. However, core files *are* produced for programs setuid to other users.

‡ See *sh*(1) for a description of the *umask* command.

§ Note that some signals, such as **SIGKILL** (in System V and 4.2BSD) and **SIGSTOP** (in 4.2BSD), cannot be caught. Moreover, on some versions of UNIX, such as Version 7, there is an inherent race condition in signal handlers, When a signal is caught, the signal trap is reset to its default value and *then* the handler is called. As a result, receiving the same signal immediately after a previous one will cause the signal to take effect regardless of whether it is being trapped. On such a version of UNIX, signals cannot be safely caught. However, if a signal is being *ignored*, sending the process a signal will *not* cause the default action to be reinstated; so, signals can be safely ignored.

```
 * This initializes the signal catching
 * vectors to call the above routine.  Note
 * any signal (including the process control
 * ones like "child just exited") will cause
 * it to be called  (You may want to change
 * that: see the text.)  If you want to allow
 * those signals which don't cause core dumps
 * to be ignored, put the code in here.
 */
initsig()
{
    register int i;        /* counter */

    /*
     * On any signal, call catcher
     * unless the signal is being ignored
     */
    for(i = 1; i < NSIG; i++)
        if (signal(i, SIG_IGN) != SIG_IGN)
            (void) signal(i, catcher);
}
```

With these two routines, catching any of **SIGQUIT**, **SIGILL**, **SIGTRAP**, **SIGIOT**, **SIGFPE**, **SIGBUS**, **SIGSEGV**, or **SIGSYS** will not cause a core dump.

Note that all of **SIGCHLD**, **SIGCONT**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU** also cause an exit. Unless there is a specific reason not to do this, this is a good idea, because if data is kept in a world-writable file, or data or lock files in a world-writable directory such as "/tmp", one can easily change information the process (presumably) relies upon. Note, however, that if the *system*(3) call is used, the **SIGCHLD** signal will be sent to the process when the command given *system* is finished; in this case, it would be wise to ignore **SIGCHLD**.

This brings us to our next point.

V. Check Data for Consistency

When writing a setuid program, it is often tempting to assume data upon which decisions are based is reliable. For example, consider a spooler. One setuid process spools jobs, and another (called the *daemon*) runs them. The daemon should not assume that the spooled jobs were spooled by the setuid program; it should try to verify this by other means, for example, checking that the owner of the command file is the same as the owner of the spooler, and that the file has not been changed since being spooled.

The precise information to be stored depends a lot on what is being done. For example, with a printing spooler, at a minimum the device number and inode number associated with each data file should be stored, since those two numbers uniquely identify any file on the system; in addition, storing the time of last modification is useful, as that will enable the daemon to determine if the data has changed since the job was spooled. All this information should be obtained *twice* – once by the spooling program, which stores it in the control file, and once again by the daemon process, which then compares it to the data stored in the control file. If *any* of the stored quantities are different, the integrity of the data file is suspect, and appropriate action should be taken. With a printing spooler, for example, the job should not print the file.

VI. Make No Assumptions About Recovery Of Errors

If the setuid program encounters an unexpected situation that the program is not prepared to handle (such as running out of file descriptors), the program should not attempt to correct for the situation. It should stop. This is the

opposite of the standard programming maxim about robustness of programs, but there is a very good reason for this rule. When a program tries to handle an unknown or unexpected situation, very often the programmer has made certain assumptions which do not hold up; for example, he may assume that lack of file descriptors means there is a problem with the system that requires the user to be given *root* privileges to fix. Such assumptions can pose extreme danger to the system and its users.

When writing a setuid program, keep track of things that can go wrong – a command too long, an input line too long, data in the wrong format, a failed system call, and so forth – and at each step ask, "if this occurred, what should be done?" If in any case the answer is "assume ...", at that point the setuid program should *stop*. Do not attempt to recover unless recovery is guaranteed; it is too easy to produce undesirable side-effects while trying to recover.

Once again, when writing a setuid program, if you are not sure how to handle a condition, exit. That way, the user cannot do any damage as a result of encountering (or creating) the condition.

For an excellent discussion of error detection and recovery under UNIX, see "Can't Happen or /* NOTREACHED */ or Real Programs Dump Core" in the *1985 Winter USENIX Proceedings* ([DARW85]).

VII. Close All But Necessary File Descriptors Before Calling exec†

This is another requirement that most setuid programs overlook. The

† *Exec* is a generic term for a number of system and library calls; these are described by the manual pages *exec*(2) in the System V manual and *execve*(2) and *execl*(3) in the 4.2 BSD manual.

problem of failing to do this becomes especially acute when the program being *exec*'ed may be a user program rather than a system one. If, for example, the setuid program were reading a sensitive file, and that file had descriptor number 9, then the user program could also read the sensitive file (because, as the manual page warns, "[d]escriptors open in the calling process remain open in the new process ...")

The easiest way to prevent this is to set a flag indicating that a sensitive file is to be closed whenever an *exec* occurs. The flag should be set immediately after opening the file. Let the sensitive file's descriptor be SENSITIVE_DESC. In both System V and 4.2 BSD, the system call

```
fcntl(SENSITIVE_DESC, F_SETFD, 1)
```

will cause the file to close across *exec*s; in both Version 7 and 4.2 BSD, the call

```
ioctl(SENSITIVE_DESC, FIOCLEX, NULL)
```

will have the same effect. (See *fcntl*(2) and *ioctl*(2) for more information.)

VIII. Reset Effective UIDs Before Calling *exec*

Resetting the effective UID and GID before calling *exec* seems obvious, but it is often overlooked. When it is, the user may find himself running a program with unexpected privileges. The following version of *system* does this:

```
/*
 * This is like system(3), but resets the
 * effective UID and GID;
 * it returns -1 if the setuid/setgid fails,
 * otherwise returns what system(3) does
 */
su_system(s)
char *s;                    /* command */
{
    /*
```

```
        * Reset the effective UID and GID
        * to the real UID and GID
        */
       if (setuid(getuid()) == -1 ||  setgid(getgid()) == -1)
           return(-1);

       /*
        * Now call system(3)
        */
       return(system(s));
}
```

IX. Check the Environment of the Process

The *environment* includes those variables which are inherited from the parent process. Among these are the variables **PATH** (which controls the order and names of directories searched by the shell for programs to be executed), **IFS** (a list of characters which are treated as word separators), and the parent's *umask*, which controls the protection mode of files that the subprocess creates. Also relevant is any attempt to restrict the process' access to the file system with the system call *chroot*(2).

The *chroot* system call, which may be used only by *root*, will force the process to treat the argument directory as the root of the file system. For example, the call

```
chroot("/usr/riacs")
```

will prevent the process from ever accessing "/usr". However, even though symbolic links are handled properly, be aware that hard links to directories outside the tree rooted at the argument directory can be followed; for example, if "/usr/demo" were linked to "/usr/riacs/demos", the sequence of system calls†

† See *chdir*(2) for more information.

```
chdir("/demos");
chdir("..")
```

would make the current working directory be "/usr" Using relative path names at this point (since an initial "/" is interpreted as "/usr/riacs"), the user could access any file on the system. Therefore, when using this call, one must be certain that no directories are linked to any of the descendants of the new root.

One of the more insidious threats comes from routines which rely on the shell to execute a program. (The routines to be wary of here are *popen*(3), *system*, *execlp*(3), and *execvp*‡.) The danger is that the shell will not execute the program intended. As an example, suppose a program that is setuid to *root* uses *popen* to execute the program *printfile*. As *popen* uses the shell to execute the command, all a user needs to do is to alter his **PATH** environment variable so that a private directory is checked before the system directories. Then, he writes his own program called *printfile* and puts it in that private directory. This private copy can do anything he likes. When the *popen* routine is executed, his private copy of *printfile* will be run, with *root* privileges!

On first blush, limiting the path to a known, safe path would seem to fix the problem. Alas, it does not. When the Bourne shell *sh* is used, there is an environment variable **IFS** which contains a list of characters that are to be treated as word separators. For example, if **IFS** is set to "e", then the shell command *spell*(1) will be treated as a command *sp* with one argument *ll* (since the "e" is treated as a blank.) Hence, one could force the setuid process to execute a program other than the one intended.

---

‡ *execlp* and *execvp* are in section 2 of the System V manual.

With a setuid program, all subprograms should be invoked by their full path name, or some path known to be safe should be prefixed to the command; and the **IFS** variable should be explicitly set to the empty string (which makes white space the only command separators.) The following version of *system* forces the path **VANILLA** to be used as the execution path for the command:

```
/*
 * This forces system(3) to use the path
 * defined in the macro VANILLA. A return of
 * -1 means there was not enough space for
 * the command and the vanilla path.
 */
#define VANILLA "/usr/ucb:/bin:/usr/bin"

safe_system(s)
char *s;                    /* command */
{
    char *cmdbuf;           /* safe path + command */

    /*
     * Allocate space for the command
     */
    cmdbuf = malloc((unsigned) (strlen(s)+strlen(VANILLA)+35));
    if (cmdbuf == NULL)
        return(-1);
    /*
     * Prepend the path to the command
     */
    (void) sprintf(cmdbuf, "IFS= ; PATH=%s ; export PATH IFS ; %s",
                   VANILLA, s);

    /*
     * Call su_system(3) so UID/GID get reset
     * (see above)
     */
    return(su_system(cmdbuf));
}
```

The danger from a badly-set *umask* is that a world-writable file owned by the effective UID of a setuid process may be produced. When a setuid process must write to a file owned by the person who is running the setuid program, and

that file must not be writable by anyone else, a subtle but nonetheless dangerous situation arises. The usual implementation is for the process to create the file, *chown*(2) it to the real UID and real GID of the process, and then write to it. However, if the *umask* is set to 0, and the process is interrupted after the file is created but before it is *chown*ed the process will leave a world-writable file owned by the user who has the effective UID of the setuid process.

There are two ways to prevent this; the first is fairly simple, but requires the effective UID to be that of *root*. (The other method does not suffer from this restriction; it is described in the next section.) The *umask*(2) system call can be used to reset the *umask* within the setuid process so that the file is at no time world-writable; this setting overrides any other, previous settings. Hence, the following routine should be used, rather than the usual *open*(2):

```
/*
 * This opens the file; it takes the same parameters as the
 * 4.2BSD and System V open(2) call; to modify for Version 7,
 * change the parameters to open(2) and this routine as appropriate.
 */
int safe_open(filename, flags, mode)
char *filename;                 /* file name */
int flags, mode;                /* how to open, creation mode */
{
    unsigned oumask;            /* old umask */
    register int opnval;        /* return value of open */

    /*
     * Reset the umask to block non-owner from
     * writing to the file.
     */
    oumask = umask(022);
    /*
     * Open the file. Note the group and world write bits
     * in the protection mask will be cleared regardless
     * of the setting of "mode", due to the umask call.
     */
    opnval = open(filename, flags, mode);
```

```
        /*
         * Restore the initial value of the umask.
         */
        (void) umask(oumask);
        return(opnval);
}
```

Upon return, the process can safely *chown* the file to the real UID and GID of the process. (Incidentally, only *root* can *chown* a file, which is why this method will not work for programs the effective UID of which is not *root*.) Note that if the process is interrupted between the *open* and the *chown* the resulting file will have the same UID and GID as the process' effective UID and GID, but the person who ran the process will not be able to write to that file (unless, of course, his UID and GID are the same as the process' effective UID and GID.)

As a related problem, *umask* is often set to a dangerous value by the parent process; for example, if a daemon is started at boot time (from the file "/etc/rc" or "/etc/rc.local"), its default *umask* will be 0. Hence, any files it creates will be created world-writable unless the protection mask used in the system call creating the file is set otherwise. The above routine will set the umask to 022 before any file is created, so it may be safely used in such situations.

Library routines should be used with great care. In particular, the routine *getlogin*(3) should not be used to determine the user's login name, since it may not return the login name expected. Rather, use *getuid*(2) and *getpwuid*(3), as the following routine does:

```
/*
 * Routine to return the login name
 * of the user of this process. If
 * none, return the UID as a string.
 * Everything is returned in a static
 * area.
```

```
 */
#include <pwd.h>

static char retval[BUFSIZ];  /* return buffer for UID */

char *glogin()
{
     register int *pwd:        /* passwd structure */

     /*
      * get the structure associated with the real UID
      */
     if ((pwd = getpwuid(getuid())) == NULL){
          /*
           * Something's out of date.
           * Return the numerical UID
           * as a string.
           */
          (void) sprintf(retval, "%d", getuid());
          return(retval);
     }
     return(pwd->pw_name);
}
```

X. Be Careful With I/O Operations

When a setuid process must write to a file owned by the person who is running the setuid program, and that file must not be writable by anyone else, a subtle but nonetheless dangerous situation arises. The usual implementation is for the process to create the file, *chown* it to the real UID and real GID of the process, and then write to it. However, if the *umask* is set to 0, and the process is interrupted after the file is created but before it is *chowned*, the process will leave a world-writable file owned by the user who has the effective UID of the setuid process.

The second method of preventing a setuid process from creating a world-writable file owned by the effective UID of the process is far more complex, but

eliminates the need for any *chown* system calls.

In this method, the process *fork*(2)s, and the child resets its effective UID and GID to the real UID and GID. The parent then writes the data to the child via *pipe*(2) rather than to the file; meanwhile, the child creates the file and copies the data from the pipe to the file. That way, the file is never owned by the user whose UID is the effective UID of the setuid process.

The following routines provide a very primitive interface for this:

```
/*
 * Routines to open, write to, and close a file;
 * this is done with a fork and pipes
 * so no chown(2)'ing need be done
 */
#include <sys/param.h>          /* may need to include <sys/types.h> */

extern int errno;               /* error code */
static int chpid;               /* child's PID */
static int ackline[NOFILE];     /* pipes for acknowledgements */
union{                          /* used to pass error of open around */
    char a[1];                  /* ... as a char array */
    int i;                      /* ... as an integer */
} u_err;

/*
 * This opens the file; it takes the same parameters as the
 * 4.2BSD and System V open(2) call; to modify for Version 7,
 * change the parameters to open(2) and this routine as appropriate.
 * The child process is contained entirely within this routine
 */
int safe_open(filename, flags, mode)
char *filename;             /* file name */
int flags, mode;            /* how to open, creation mode */
{
    int desc[2];            /* pipe for information flow */
    int status[2];          /* pipe for acknowledgement */
    int forkval;            /* value returned from fork(2) */

    /*
     * Build the pipes.
     * Information to be written to the file
     * flows through desc to the child.
```

```
 * The status pipe carries acknowledgements
 * from the child to the parent.
 */
if (pipe(desc) == -1)
    return(-1);
if (pipe(status) == -1){
    (void) close(desc[0]);
    (void) close(desc[1]);
    return(-1);
}

/*
 * Spawn the child process.
 */
if ((forkval = fork()) == -1){
    (void) close(desc[0]);
    (void) close(desc[1]);
    (void) close(status[0]);
    (void) close(status[1]);
    return(-1);
}
else if (forkval == 0){
    /*
     * This is the child: it never leaves this
     * branch of the conditional.
     * First, some useful variables.
     */
    char buf[BUFSIZ];      /* I/O buffer */
    int fildes = -1;       /* descriptor of output file */
    int ctread;            /* count of bytes read */

    /*
     * Reset effective UID, GID.
     */
    if (setuid(getuid()) < 0 || setgid(getgid()) < 0)
        _exit(1);

    /*
     * Read only from the desc pipe,
     * and write only to the status pipe.
     */
    (void) close(desc[1]);
    (void) close(status[0]);

    /*
     * Open the file as requested.
     * Handle an error by exiting.
```

```
         */
        if ((fildes = open(filename, flags, mode)) < 0)
            /*
             * Shucks ... pass back the error number.
             */
            u_err.i = errno;
            (void) write(status[1], u_err.a, sizeof(int));
            _exit(0);
        }
        /*
         * Signal all's well.
         */
        u_err.i = -1;
        (void) write(status[1], u_err.a, sizeof(int));
        /*
         * Main loop -- just read from the desc pipe
         * until there's nothing more to read.
         * Do acknowledge every read, though.
         */
        u_err.i = -1;
        while((ctread = read(desc[0], buf, BUFSIZ)) > 0){
            if (write(fildes, buf, ctread) != ctread){
                u_err.i = errno;
                (void) write(status[1], u_err.a, sizeof(int));
                _exit(0);
            }
            (void) write(status[1], u_err.a, sizeof(int));
        }
        /*
         * We just read an end of file.
         * Close the pipe and the file and quit.
         */
        (void) close(status[1]);
        (void) close(desc[0]);
        (void) close(fildes);
        _exit(0);
    }
    /*
     * This is the parent process.
     * Close the descriptors we don't need.
     */
    (void) close(status[1]);
    (void) close(desc[0]);
    /*
     * Now save the status descriptor.
     */
    ackline[desc[1]] = status[0];
```

```
    /*
     * Get the status of the open(2).
     */
    if (read(status[0], u_err.a, sizeof(int)) != sizeof(int)){
        /*
         * No status was sent - assume catastrophe.
         */
        (void) close(status[0]);
        (void) close(desc[1]);
        return(-1);
    }
    /*
     * We read something and it wasn't good, so
     * set errno to the error code and quit.
     */
    if (u_err.i != -1){
        (void) close(status[0]);
        (void) close(desc[1]);
        errno = u_err.i;
        return(-1);
    }
    /*
     * Return the pipe descriptor.
     */
    return(desc[1]);
}

/*
 * This writes to the child/file and
 * takes the same parameters as write(2).
 */
int safe_write(fd, buf, bufsiz)
int fd;                 /* file descriptor from safe_open */
char *buf;              /* data to be written */
int bufsiz;             /* number of bytes to be written */
{
    register int i;         /* counter in a for loop */
    register int tokid;     /* bytes written to child */

    /*
     * Do this in packets of BUFSIZ
     * so you don't flood the pipe.
     */
    for(i = 0; i < bufsiz; i += BUFSIZ){
        /*
         * See how much to write.
         */
```

```
            min = bufsiz - i;
            if (min > BUFSIZ)
                min = BUFSIZ;
            /*
             * Write it.
             */
            if (write(fd, buf, min) != min)
                return(i);
            /*
             * Wait for an acknowledgement;
             * if none, assume the worst.
             */
            if (read(ackline[fd], u_err.a, sizeof(int)) != sizeof(int))
                return(-1);
            if (u_err.i != -1){
                errno = u_err.i;
                return(-1);
            }
        }
    }

/*
 * This closes the child/file and
 * takes the same parameters as close(2).
 * Note it waits for the child process.
 */
int safe_close(fd)
int fd;                     /* file descriptor from safe_open */
{
    register int waitval;           /* process that died */

    /*
     * Close the send pipe and the acknowledgement pipe.
     */
    (void) close(fd);
    (void) close(ackline[fd]);
    /*
     * Wait for the child to bite the big one.
     */
    while((waitval = wait(0)) != -1 && waitval != chpid);
}
```

Conclusion

To summarize, the rules to remember when writing a setuid program are:

- be as restrictive as possible in choosing the UID
- do not write setuid shell scripts
- do not use *creat* for locking
- catch all signals
- check data for consistency
- make no assumptions about recovery of errors
- close all but necessary file descriptors before calling *exec*
- reset effective UIDs before calling *exec*
- check the environment of the process
- be careful with I/O operations

Setuid programs explicitly violate the protection scheme designed into UNIX. On systems where security is not a problem, this is a blessing, since it enables many things to be done easily that otherwise would be very difficult; but on systems where security is a problem, these programs also pose very real threats. Unfortunately, they are also very necessary, so the designers and implementors of setuid programs should take great care when writing them.

Acknowledgements: Thanks to Bob Brown, Peter Denning, George Gobel, Chris Kent, Rich Kulawiec, Dawn Maneval, and Kirk Smith, who reviewed an earlier draft of this paper, and made many constructive suggestions.

References

[ALEP71] Aleph-Null, "Computer Recreations," *Software – Practise and Experience* 1(2) pp. 201 – 204 (April – June 1971)

[DARW85] Darwin, Ian and Collyer, Geoff, "Can't Happen or /* NOTREACHED */ or Real Programs Dump Core," 1985 Winter USENIX Proceedings (January 1985)

[LAMP73] Lampson, Butler, "A Note on the Confinement Problem," *CACM* **16**(10) pp. 613 – 615 (October 1973)

[TRUS80] Truscott, Tom and Ellis, James, "On the Correctness of Set-User-ID Programs," Department of Computer Science, Duke University (unpublished)

[UPM83] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (August 1983)

[UPM84] *UNIX Programmer's Manual, Version 1.0*, Silicon Graphics, Inc., Mountain View, CA (June 1984)